Formal Derivation of Pointer Algorithms

Bernhard Möller

Abstract We show that the well-known unfold/fold transformation strategy also is fruitful for the (formal) derivation of correct pointer algorithms. The key that allows this extension is the algebra of partial maps which allows convenient description and manipulation of pointer structures at the functional level.

1 Introduction

It is well-known that algorithms involving pointers are both difficult to write and to verify. The reason is that, due to the implict connections through paths within a pointer structure, the side effects of a pointer assignment are usually much harder to survey than those of an ordinary assignment. With this paper we want to show that these difficulties can be greatly reduced by making the store, which is an implicit global parameter in procedural languages, into an explicit parameter and by passing to an applicative treatment using a suitable algebra of operations on the store.

The storage state of a von Neumann machine can be viewed as a total mapping from addresses to certain values. A part of such a state that forms a logical unit may then be represented by a partial submapping of that mapping. This gives the possibility of describing the state in a modularized way as the union of the submappings for its logical subunits. In the case of pointer structures this means that the usual "spaghetti" structure of the complete state can be (at least partly) disentangled. Therefore we use the algebra of partial maps as our tool for specifying and developing pointer algorithms in a formal and yet convenient way.

We restrict ourselves here to the case of singly linked lists. However, the approach is not limited to such simple structures: In [3] we have derived an efficient and intricate garbage collection algorithm for a storage structure that allows the representation of arbitrary graphs.

2 The Algebra of Partial Maps

The use of algebraic operations on maps for describing the effect of a program dates back at least to [11]. The most useful operation in our setting, viz. map union, however, seems to have been neglected until recently [3,10].

A **(partial) map** m from a set M to a set N is a subset of $M \times N$ such that $(x, y) \in m \wedge (x, z) \in m \;\Rightarrow\; y \equiv z$. Some of our notation derives from this set view of maps. E.g., by $\emptyset$ we denote the empty partial map from M to N. For finite maps we assume a boolean-valued equality test $=$. This is to be distinguished from the semantic equivalence $\equiv$ of expressions: we have

$$m = n \equiv \mathsf{true} \;\Leftrightarrow\; m \equiv n\ .$$

Let $m : P \to Q$ be a partial map. We write $\downarrow m, \uparrow m$ for **domain** and **range** of m, resp. Moreover, we define

$$set(m) \overset{\text{def}}{\equiv} \downarrow m \;\cup \uparrow m\ .$$

For $s \subseteq P$, we denote by $s{\uparrow}m$ the **image** of s under m. Likewise, $t{\downarrow}m$ is the **inverse image** of $t \subseteq Q$ under m. Finally, $[s \mapsto y]$ is the constant map $\{(x, y) \,|\, x \in s\}$. In using this notation we omit singleton set braces, i.e., we write $x{\uparrow}m, y{\downarrow}m, [x \mapsto y]$ instead of $\{x\}{\uparrow}m, \{y\}{\downarrow}m, [\{x\} \mapsto y]$. Note that $[x \mapsto y] \equiv \{(x, y)\}$. To cope with partialities in an algebraically convenient way, we define, for maps m, n and elements $x, y \in P$,

$$[m(x) \mapsto n(y)] \overset{\text{def}}{\equiv} \emptyset$$

if $x \notin \downarrow m$ or $y \notin \downarrow n$. Also, we set

$$m(x) \equiv n(y) \overset{\text{def}}{\Leftrightarrow} x{\uparrow}m \equiv y{\uparrow}n\ .$$

This means that $m(x), n(y)$ are both undefined or both defined and equal.

The **restriction** of a map $m : M \to N$ to a set $s \subseteq M$ is

$$m|s \overset{\text{def}}{\equiv} m \cap (s \times N)\ .$$

Moreover,

$$m \ominus s \overset{\text{def}}{\equiv} m|\overline{s}.$$

Here again we omit singleton set braces, i.e., we write $m \ominus x$ instead of $m \ominus \{x\}$. Note that both $m|s \subseteq m$ and $m \ominus s \subseteq m$. The following decomposition property is the key to recursions over maps:

$$m \equiv m|s \cup m \ominus s\ .$$

Two maps $m, n : M \to N$ are **compatible** if $m|(\downarrow m \cap \downarrow n) \equiv n|(\downarrow m \cap \downarrow n)$. This holds in particular if $\downarrow m \cap \downarrow n \equiv \emptyset$. For compatible m, n their union $m \cup n$ is again a map. This generalizes to families $(m_i)_{i \in I}$ of maps (I may even be infinite) if the maps m_i are pairwise compatible; we then write $\bigcup\limits_{i \in I} m_i$ for the union map. If $I \equiv \emptyset$, we set $\bigcup\limits_{i \in I} m_i \equiv \emptyset$ as well. It should be clear that $\emptyset$, $[. \mapsto .]$, and $\bigcup$ form a complete set of constructors for the set of partial maps, since we have

$$m \equiv \bigcup_{x \in \downarrow m} [x \mapsto m(x)] \ .$$

The operation of map union is the key tool in obtaining a modular description of pointer structures, since it allows viewing a (total) storage state as the union of those of its (partial) substates that form logical units. This aspect of modularization is reflected by a large number of distributive laws that allow propagation of operations to substates of a state. For the operations introduced so far we have:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
s\downarrow(m \cup n) & \equiv & s\downarrow m \ \cup s\downarrow n & t\uparrow(m \cup n) & \equiv & t\uparrow m \cup t\uparrow n \\
\downarrow(m \cup n) & \equiv & \downarrow m \ \cup \downarrow n & \uparrow(m \cup n) & \equiv & \uparrow m \cup \uparrow n \\
(m \cup n)|s & \equiv & m|s \cup n|s & (m \cup n) \ominus s & \equiv & m \ominus s \cup m \ominus s \ .
\end{array}$$

Another important operation is **map overwriting** (see e.g. [6]): Given maps $m, n : M \to N$ we define

$$m \mathrel{+\!\!\!<} n \overset{\text{def}}{\equiv} (m \ominus \downarrow n) \ \cup \ n \ .$$

Hence,

$$(m \mathrel{+\!\!\!<} n)(x) \equiv \mathsf{if}\ x \in \ \downarrow n\ \mathsf{then}\ n(x)\ \mathsf{else}\ m(x)\ \mathsf{fi}.$$

In other words, $m \mathrel{+\!\!\!<} n$ results from m by changing the values according to the prescription of n (if any). For example, $m \mathrel{+\!\!\!<} [x \mapsto y]$ sets the value of x to y. This operation will be our main tool for describing selective updating. Its most important properties for our purposes are the following ones:

1. Monoid properties:
   $\emptyset \mathrel{+\!\!\!<} m \equiv m \mathrel{+\!\!\!<} \emptyset \equiv m$
   $(l \mathrel{+\!\!\!<} m) \mathrel{+\!\!\!<} n \equiv l \mathrel{+\!\!\!<} (m \mathrel{+\!\!\!<} n)$
2. Overwriting and union:
   $m \mathrel{+\!\!\!<} n \equiv n \mathrel{+\!\!\!<} m$ iff m and n are compatible.
   In this case, $m \mathrel{+\!\!\!<} n \equiv m \cup n$.

3. Domain properties:
   $\downarrow(m \mathbin{\lhd\!\!\!+} n) \equiv \downarrow m \cup \downarrow n$
   $m \mathbin{\lhd\!\!\!+} n \equiv n \Leftrightarrow \downarrow m \subseteq \downarrow n$
4. Overwriting and submaps:
   $m \mathbin{\lhd\!\!\!+} n \equiv m \Leftrightarrow n \subseteq m$
5. Sequentialization:
   $l \mathbin{\lhd\!\!\!+} (m \cup n) \equiv (l \mathbin{\lhd\!\!\!+} m) \mathbin{\lhd\!\!\!+} n$
   provided m and n are compatible.
6. Annihilation:
   $m \subseteq l \Rightarrow l \mathbin{\lhd\!\!\!+} (m \cup n) \equiv l \mathbin{\lhd\!\!\!+} n$
   provided m and n are compatible. This is an immediate consequence of the sequentialization and submap properties.
7. Distributivity:
   $(l \cup m) \mathbin{\lhd\!\!\!+} n \equiv (l \mathbin{\lhd\!\!\!+} n) \cup (m \mathbin{\lhd\!\!\!+} n)$
   provided l and m are compatible.
8. Localization:
   $\downarrow l \cap \downarrow n \equiv \emptyset \Rightarrow (l \cup m) \mathbin{\lhd\!\!\!+} n \equiv l \cup (m \mathbin{\lhd\!\!\!+} n)$
   provided l and m are compatible. This property allows localizing side effects to that part of a store they really affect.

A thorough discussion of the properties of $\mathbin{\lhd\!\!\!+}$ can be found in [5].

A map m is **injective** if $\forall\, x, y \in \downarrow m : m(x) = m(y) \Rightarrow x = y$. Then its **inverse** m^{-1} is given by $m^{-1} \equiv \bigcup_{x \in \downarrow m} [m(x) \mapsto x]$ and satisfies

$$\begin{array}{rcl@{\qquad\qquad}rcl} s{\downarrow}m^{-1} & \equiv & s{\uparrow}m & t{\uparrow}m^{-1} & \equiv & t{\downarrow}m \\ \downarrow m^{-1} & \equiv & \uparrow m & \uparrow m^{-1} & \equiv & \downarrow m\ . \end{array}$$

Moreover,

$$(m \cup n)^{-1} \equiv m^{-1} \cup n^{-1}$$

provided $m \cup n$ is injective.

The map operations introduced enjoy a vast number of further useful algebraic laws. Some of them can be found in [3].

3 Chains

3.1 Basic Notions

As an example of how to describe pointer structures within the algebra of maps we now study singly linked lists. We abstract from the concrete contents of the records in such a list and consider only their interrelationship through the pointers, since this is the only source of problems in pointer algorithms. Then a **state** simply is a finite partial map $m : \mathsf{cell} \to \mathsf{cell}$ where cell is the set of storage cells; the set of states is denoted by state.

The idea of following the pointers in a singly linked list within a state m is captured by considering m as a relation and passing to its transitive closure m^+: For $x, y \in set(m)$,

$$x\, m^+\, y \overset{\text{def}}{\equiv} y \in \bigcup_{i \in \mathbb{N}\setminus\{0\}} x{\uparrow}m^i$$

where

$$\begin{aligned} m^0 &\overset{\text{def}}{\equiv} id_{set(m)} \\ m^{i+1} &\overset{\text{def}}{\equiv} m \circ m^i \ , \end{aligned}$$

and $\circ$ denotes the usual composition of partial maps. Hence $x\, m^+\, y$ iff y can be reached from x following the links of m (at least once). We also need the reflexive transitive closure m^* of m given by

$$x\, m^*\, y \overset{\text{def}}{\equiv} y \in \bigcup_{i \in \mathbb{N}} x{\uparrow}m^i \ .$$

We call m a **chain**, if m^+ is a linear strict-order on $set(m)$, i.e., iff the predicate $ischain(m)$ holds where

$$\begin{array}{lcll} ischain(m) & \overset{\text{def}}{\equiv} & \forall\, x, y, z \in set(m) : & \\ & & \neg\, x\, m^+\, x & \text{(irreflexivity)} \\ & \wedge & x \neq y \Rightarrow (x\, m^+\, y \vee y\, m^+\, x) & \text{(linearity).} \end{array}$$

Irreflexivity excludes the existence of cycles within the list, whereas linearity implies that the list is connected, i.e, that, given two distinct cells in the list, one of them can be reached from the other following the links of the list.

Lemma 3.1 *Let m be a chain.*
(1) *m is injective.*
(2) *Assume $x, y \in set(m)$. If $x\, m^+\, y$ then $\neg\, y\, m^+\, x$.*

Proof.
(1) Let $m(x) \equiv m(y) \equiv z$ and assume $x \not\equiv y$. Then w.l.o.g. $x\, m^+\, y$. Since m is a map, it follows that $x\, m\, z\, m^*\, y\, m\, z$, i.e., $z\, m^+\, z$, a contradiction.
(2) Assume $x\, m^+\, y$ and $y\, m^+\, x$. By transitivity then $x\, m^+\, x$ contradicting the irreflexivity of m^+.

$\square$

Property (1) means that each cell in a chain is referred to by at most one cell, i.e., absence of sharing, whereas (2) again means absence of cycles.

The following lemma states a property that is very useful for treating combinations of chains:

Lemma 3.2 *Let m and n be maps such that $\uparrow n \cap \downarrow m \equiv \emptyset$. Then $(m \cup n)^+ \equiv m^+ \cup m^+ n^+ \cup n^+$, where $m^+ n^+$ is the relational product of m^+ and n^+ (in diagrammatic order).*

Proof. From $\uparrow n \cap \downarrow m \equiv \emptyset$ it follows that $n\, m \equiv \emptyset$. Now a straightforward induction shows

$$(m \cup n)^i \equiv \bigcup_{k=0}^{i} m^{i-k}\, n^k \ .$$

From this the claim is immediate. $\square$

Chains are usually parts of a larger state and are distinguished by header cells. Hence we define, for arbitrary $m : \mathsf{state}$ and $x \in \mathsf{cell}$,

$$\mathit{from}(x, m) \overset{\text{def}}{\equiv} m|\{y \mid x\, m^*\, y\} \quad (\equiv m|(\bigcup_{i \in \mathbb{N}} x {\uparrow} m^j)) \ .$$

Then $\mathit{from}(x, m)$ is the linked list within m that starts from x (inclusive of x), if any. Note that this list need not be a chain, since it may contain a cycle.

Lemma 3.3

(1) $\mathit{from}(x, m) \equiv \bigcup_{i \in \mathbb{N}} [m^i(x) \mapsto m^{i+1}(x)]$.

(2) $x \notin \downarrow m \Rightarrow \mathit{from}(x, m) \equiv \emptyset$.

Proof.

(1) $\mathit{from}(x, m) \equiv m|(\bigcup_{i \in \mathbb{N}} x {\uparrow} m^i) \equiv \bigcup_{i \in \mathbb{N}} m|(x {\uparrow} m^i) \equiv$
$\bigcup_{i \in \mathbb{N}} [m^i(x) \mapsto m(m^i(x))] \equiv \bigcup_{i \in \mathbb{N}} [m^i(r) \mapsto m^{i+1}(r)]$.

(2) is immediate from the definition.

$\square$

As a first example for reasoning about linked lists we now prove that overwriting at the beginning of a list does not influence the rest of the list, provided the list itself is not a cycle:

Lemma 3.4 *Assume that $\neg\, x\, m^+\, x$. Then for arbitrary y we have*

$$\mathit{from}(m(x), m) \equiv \mathit{from}(m(x), m \Leftarrow [x \mapsto y]) \ .$$

Proof. We show by induction on $i \in \mathbb{N}$ that $(m \Lleftarrow [x \mapsto y])^i \equiv m^i(z)$ for all z such that $x\, m^+\, z$. The base case $i \equiv 0$ is trivial. Assume now that the assertion holds for i and consider an arbitrary z with $x\, m^+\, z$. Since $\neg\, x\, m^+\, x$ we know $z \not\equiv x$. Hence $(m \Lleftarrow [x \mapsto y])(z) \equiv m(z)$. Moreover, $x\, m^+\, m(z)$ provided $m(z)$ is defined. Now

$$\begin{aligned}
&(m \Lleftarrow [x \mapsto y])^{i+1}(z)\\
\equiv\ &(m \Lleftarrow [x \mapsto y])^{i}((m \Lleftarrow [x \mapsto y])(z))\\
\equiv\ &(m \Lleftarrow [x \mapsto y])^{i}(m(z))\\
\equiv\ &\text{(by induction hypothesis resp. undefinedness)}\\
&m^i(m(z)\\
\equiv\ &m^{i+1}(z).
\end{aligned}$$

This concludes the induction. Now we have

$$\begin{aligned}
&\mathit{from}(m(x), m)\\
\equiv\ &\text{(by the previous lemma)}\\
&\bigcup_{i\in\mathbb{N}} [m^i(m(x)) \mapsto m^{i+1}(m(x))]\\
\equiv\ &\bigcup_{i\in\mathbb{N}} [(m \Lleftarrow [x \mapsto y])^i(m(x)) \mapsto (m \Lleftarrow [x \mapsto y])^{i+1}(m(x))]\\
\equiv\ &\mathit{from}(m(x), m \Lleftarrow [x \mapsto y]).
\end{aligned}$$

□

This property will be useful in reversing a chain.

3.2 Anchored Chains

We now turn to the special case of terminating chains. These are cycle-free singly linked lists in which each element has only finitely many successors. Formally, a chain m is **terminating** if $m \equiv 0$ or $\mathit{set}(m)$ contains a greatest element $\mathit{last}(m)$ w.r.t. m^+.

Lemma 3.5 *Let $m \not\equiv \emptyset$ be a terminating chain. Then*
(1) $\downarrow\! m \equiv \mathit{set}(m) \backslash \{\mathit{last}(m)\}$.
(2) $\uparrow\! m \subseteq\, \downarrow\! m \cup \{\mathit{last}(m)\}$.
(3) $\uparrow\! m \backslash \downarrow\! m \equiv \{\mathit{last}(m)\}$.

Proof.
(1) ($\subseteq$) $\downarrow\! m \subseteq \mathit{set}(m)$ holds by definition. Assume $\mathit{last}(m) \in\, \downarrow\! m$ and let $y \equiv m(\mathit{last}(m))$. Then $y \in \mathit{set}(m)$ and $\mathit{last}(m)\, m^+\, y \not\equiv \mathit{last}(m)$, contradicting the fact that $\mathit{last}(m)$ is the greatest element of $\mathit{set}(m)$.
($\supseteq$) Let $x \in \mathit{set}(m)\backslash\{\mathit{last}(m)\}$. Then $x\, m^+\, \mathit{last}(m)$, since $\mathit{last}(m)$ is the greatest element of $\mathit{set}(m)$. But then $x \in\, \downarrow\! m$, since otherwise all sets $x{\uparrow} m^i$ $(i > 0)$ would be empty.

(2) By definition, $\uparrow m \subseteq set(m)$. But by (1), $set(m) \equiv \downarrow m \cup \{last(m)\}$.
(3) Using (1) we get
$\uparrow m \backslash \downarrow m \equiv \uparrow m \backslash (set(m) \backslash \{last(m)\}) \equiv$
$\uparrow m \backslash set(m) \cup \uparrow m \cap \{last(m)\} \equiv \{last(m)\})$.

$\square$

Frequently one uses a special chain terminator common to all chains considered (e.g., nil in Pascal). Let therefore $\square \in \mathsf{cell}$ be a distinguished element, called the **anchor**. The elements of $\mathsf{cell} \backslash \{\square\}$ are called **proper cells**. In the sequel we require $\square \notin \downarrow m$ for all states m considered.

We define the predicate

$$\begin{array}{lcl} isanchored(m) & \overset{\text{def}}{\equiv} & ischain(m) \wedge \\ & & (m = \emptyset \curlyvee last(m) = \square) \ , \end{array}$$

where $\curlyvee$ is the sequential or conditional disjunction evaluated from left to right. Thus, a nonempty chain is anchored iff it is terminated by $\square$.

For a nonempty anchored chain the last element but one is the last "proper" element. Therefore we define, for anchored m,

$$\{lbo(m)\} \overset{\text{def}}{\equiv} \mathsf{if}\ m = \emptyset\ \mathsf{then}\ \{\square\}\ \mathsf{else}\ \square \downarrow m\ \mathsf{fi}\ .$$

3.3 Well-Founded Chains

Dually to terminating chains we also consider chains with starting elements. Call a chain m **well-founded** if $m \equiv \emptyset$ or $set(m)$ contains a least element $first(m)$ w.r.t. m^+.

Lemma 3.6 *Let $m \not\equiv \emptyset$ be a well-founded chain. Then*
(1) $\uparrow m \equiv set(m) \backslash \{first(m)\}$.
(2) $\downarrow m \subseteq \uparrow m \cup \{first(m)\}$.
(3) $\downarrow m \backslash \uparrow m \equiv \{first(m)\}$.

Proof. Dual to that of the previous lemma. $\square$

We set, for well-founded $m \not\equiv \emptyset$,

$$rest(m) \equiv m \ominus first(m)\ .$$

These operations again show useful properties:

Lemma 3.7 *Let $m \not\equiv \emptyset$ be a well-founded chain. Then*
(1) $m \equiv [first(m) \mapsto m(first(m))] \cup rest(m)$.

(2) $\mathit{rest}(m) \not\equiv \emptyset \Rightarrow \mathit{first}(\mathit{rest}(m)) \equiv m(\mathit{first}(m))$.
(3) *If* m *is also anchored,* $\mathit{rest}(m) \equiv \emptyset \Leftrightarrow m(\mathit{first}(m)) \equiv \square$.

Proof.
(1) is immediate from the decomposition property of maps.
(2) $\{\mathit{first}(\mathit{rest}(m))\} \equiv {\downarrow}\mathit{rest}(m) \backslash {\uparrow}\mathit{rest}(m) \equiv$
$({\downarrow}m \backslash \{\mathit{first}(m)\}) \cup ({\uparrow}m \backslash \{m(first(m)\}) \equiv$
$({\downarrow}m \backslash \{\mathit{first}(m)\} \backslash {\uparrow}m) \cup ({\downarrow}m \backslash \{\mathit{first}(m)\} \cap \{m(\mathit{first}(m))\}) \equiv$
$({\downarrow}m \backslash {\uparrow}m \backslash \{\mathit{first}(m)\}) \cup \{m(\mathit{first}(m))\} \equiv$
$(\{\mathit{first}(m)\} \backslash \{\mathit{first}(m)\}) \cup \{m(\mathit{first}(m))\} \equiv \{m(\mathit{first}(m))\}$.
(3)($\Rightarrow$) By (1), $m \equiv [\mathit{first}(m) \mapsto m(\mathit{first}(m))]$. Hence $\square \equiv \mathit{last}(m) \equiv m(\mathit{first}(m))$.
($\Leftarrow$) We have $\mathit{set}(m) \equiv \bigcup_{i \in \mathbb{N}} \mathit{first}(m){\uparrow}m^i$. However, $\mathit{first}(m){\uparrow}m \equiv \{\square\}$ implies $\mathit{first}(m){\uparrow}m^i \equiv \emptyset$ for $i > 2$, since $\square \not\in {\downarrow}m$. Hence we have $\mathit{set}(m) \equiv \{\mathit{first}(m), \square\}$ and $m \equiv [\mathit{first}(m) \mapsto \square]$. Therefore $\mathit{rest}(m) \equiv \emptyset$.

$\square$

Corollary 3.8 *If* $\mathit{from}(x, m) \not\equiv \emptyset$ *is a well-founded chain, we have*

$$\mathit{first}(\mathit{from}(x, m)) \equiv x \text{ and } \mathit{rest}(\mathit{from}(x, m)) \equiv \mathit{from}(m(x), m) \ .$$

If, moreover, $\mathit{from}(x, m)$ *is anchored,*

$$from(x, m) \equiv \mathsf{if}\ m(x) = \square\ \mathsf{then}\ [x \mapsto \square]\ \mathsf{else}\ [x \mapsto m(x)] \cup \mathit{from}(m(x), m)\ \mathsf{fi}\ .$$

It can be shown that a chain is both well-founded and terminating iff it is finite as a set of argument/value pairs. For finite anchored chains we now want to derive a recursive version of the function *lbo* defined in the previous section. Assume $m \not\equiv \emptyset$. We calculate

$$
\begin{aligned}
& \{\mathit{lbo}(m)\} \\
\equiv\ & \square{\downarrow}m \\
\equiv\ & \text{(by Lemma 3.7(1))} \\
& \square{\downarrow}([\mathit{first}(m) \mapsto m(\mathit{first}(m))] \cup \mathit{rest}(m)) \\
\equiv\ & \square{\downarrow}[\mathit{first}(m) \mapsto m(\mathit{first}(m))] \cup \square{\downarrow}\mathit{rest}(m) \\
\equiv\ & \mathsf{if}\ m(\mathit{first}(m)) = \square\ \mathsf{then}\ \square{\downarrow}[\mathit{first}(m) \mapsto m(\mathit{first}(m))] \cup \square{\downarrow}\mathit{rest}(m) \\
& \qquad\qquad\qquad\qquad\quad \mathsf{else}\ \ \square{\downarrow}[\mathit{first}(m) \mapsto m(\mathit{first}(m))] \cup \square{\downarrow}\mathit{rest}(m)\ \mathsf{fi} \\
\equiv\ & \text{(definition, Lemma 3.7(3))} \\
& \mathsf{if}\ m(\mathit{first}(m)) = \square\ \mathsf{then}\ \{\mathit{first}(m)\} \cup \square{\downarrow}\emptyset \\
& \qquad\qquad\qquad\qquad\quad \mathsf{else}\ \ \emptyset \cup \square{\downarrow}\mathit{rest}(m)\ \mathsf{fi}
\end{aligned}
$$

$\equiv$ (simplification, fold *lbo*)
$\quad$ $\mathsf{if}\ m(\mathit{first}(m)) = \square\ \mathsf{then}\ \{\mathit{first}(m)\}$
$\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \ \{lbo(rest(m))\}\ \mathsf{fi}\ .$

Hence, for $m \not\equiv \emptyset$,

$$lbo(m)\ \equiv\ \mathsf{if}\ m(\mathit{first}(mn)) = \square\ \mathsf{then}\ \mathit{first}(m)\,\mathsf{else}\,lbo(rest(m))\ \mathsf{fi}\ .$$

Termination of this recursion is obvious.

4 Concatenation of Chains

In the sequel, concerning semantics and notation we closely follow the language CIP-L [1,2]. In addition, we use the notation

$$B\ \rhd\ E$$

as an abbreviation for the expression

$$\mathsf{if}\ B\ \mathsf{then}\ E\,\mathsf{else}\,\mathsf{error}\ \mathsf{fi}$$

which is equivalent to E if the precondition B is satisfied and undefined otherwise. See [9] for a more detailed discussion of this construct. An analogous construction applies to statements.

4.1 Specification and First Explicit Solution

We now want to specify and develop an algorithm for concatenating two non-overlapping anchored chains "in situ". (The case of empty chains is trivial and would only lead to tedious case distinctions.) As an auxiliary notion we need the restriction of a relation R to a set s, viz.

$$R||s\ \stackrel{\text{def}}{\equiv}\ R \cap s \times s\ .$$

Our specification then reads

$$\begin{array}{rcl}
concpc(m,n) & \stackrel{\text{def}}{\equiv} & m \neq \emptyset\ \wedge\ n \neq \emptyset\ \mathrel{\dot\wedge} \\
 & & isanchored(m)\ \wedge\ isanchored(n)\ \wedge\ {\downarrow}m\ \cap\ {\downarrow}n = \emptyset \\
conc(m,n) & \stackrel{\text{def}}{\equiv} & concpc(m,n)\ \rhd\ \mathsf{some\,state}\ l: \\
 & (1) & isanchored(l) \wedge \\
 & (2) & {\downarrow}l = {\downarrow}m \cup {\downarrow}n \wedge \\
 & (3) & l^+||{\downarrow}m = m^+||{\downarrow}m\ \wedge\ l^+||{\downarrow}n = n^+||{\downarrow}n \wedge \\
 & (4) & \forall\, x \in {\downarrow}m : \forall\, y \in {\downarrow}n : x\, l^+\, y\ .
\end{array}$$

Let us explain the particular form of the specification. It is an attempt to characterize the concatenation of two chains solely in terms of the orderings they induce on the cells involved. The precondition *concpc* requires the inputs to be anchored chains the sets of proper cells of which do not overlap. (1) states that the result should again be an anchored chain. (2) actually is the requirement that the concatenation should be performed “in situ”, since it stipulates that the proper cells of the result chain should be exactly the same as those of the component cells; no “new” cells may be used. (3) states that the ordering of the cells within the component chains must not be disturbed by the concatenation. (4), finally, requires all proper cells of the first argument to occur in the concatenation before all proper cells of the second argument.

To obtain an explicit version of cone, draw some conclusions from the specification, i.e., we derive necessary conditions for a map to be a solution of our problem. The hope is to derive more explicit conditions than the ones given.

Requirement (4) implies $\mathit{lbo}(m)\, l^+ \mathit{first}(n)$. However, since by (2) and (3) $\mathit{lbo}(m)$ is maximal w.r.t. l^+ in $\downarrow m$ and $\mathit{first}(n)$ is minimal in $\downarrow n$ w.r.t. l^+ , there cannot be any element $z \in \downarrow m \cup \downarrow n$ with $\mathit{lbo}(m)\, l^+ z\, l^+ \mathit{first}(n)$, so that we must have $\mathit{lbo}(m)\, l\, \mathit{first}(n)$, i.e., $[\mathit{lbo}(m) \mapsto \mathit{first}(n)] \subseteq l$. Moreover, (2) and (3) require that l should be “almost” the union of m and n. Since, however, $m(\mathit{lbo}(m)) \equiv \Box$, we need to adjust m at that point. This suggests the choice

$$l \equiv (m \mathbin{\leftarrow\!\!\!+} [\mathit{lbo}(m) \mapsto \mathit{first}(n)]) \cup n.$$

This is a version many a reader probably would have written down immediately without bothering to come up with a specification like our original one. However, the explicit version does not in any way reflect that concatenation of two chains “in situ” is intended; it merely corresponds to an assignment that redirects one pointer. Therefore we think that our implicit original specification is more adequate as far as understanding of the task is concerned; the price we have to pay is that we now need to verify that the explicit version actually meets the implicit specification. Fortunately, it turns out that this is not too hard using the algebraic properties we have at hand.

For abbreviation we set

$$p \overset{\text{def}}{\equiv} m \mathbin{\leftarrow\!\!\!+} [\mathit{lbo}(m) \mapsto \mathit{first}(n)]\ .$$

We have

$$\begin{aligned} & \uparrow n \cap \downarrow p \\ \equiv\ & \uparrow n \cap \downarrow m \end{aligned}$$

$\equiv (\downarrow n \backslash \mathit{first}(n) \cup \{\Box\}) \cap \downarrow m$
$\equiv \emptyset$.
Therefore Lemma 3.2 applies and we have

$$l^+ \equiv p^+ \cup p^+ n^+ \cup n^+ \ .$$

Furthermore, $p \equiv m \ominus \mathit{lbo}(m) \cup [\mathit{lbo}(m) \mapsto \mathit{first}(n)]$ and
$\uparrow[\mathit{lbo}(m) \mapsto \mathit{first}(n)] \cap \downarrow m \ominus \mathit{lbo}(m)$
$\equiv \uparrow\{\mathit{first}(n)\} \cap (\downarrow m \backslash \{\mathit{lbo}(m)\})$
$\equiv \emptyset$,
so that, again by Lemma 3.2,
p^+
$\equiv (m \ominus \mathit{lbo}(m))^+ \cup$
$(m \ominus \mathit{lbo}(m))^+ \, [\mathit{lbo}(m) \mapsto \mathit{first}(n)]^+ \cup$
$[\mathit{lbo}(m) \mapsto \mathit{first}(n)]^+$
$\equiv (m \ominus \mathit{lbo}(m))^+ \cup$
$(m \ominus \mathit{lbo}(m))^+ \, [\mathit{lbo}(m) \mapsto \mathit{first}(n)] \cup$
$[\mathit{lbo}(m) \mapsto \mathit{first}(n)]$
$\equiv (m \ominus \mathit{lbo}(m))^+ \cup$
$[\downarrow m \mapsto \mathit{first}(n)]$.
Hence
l^+
$\equiv p^+ \cup p^+ n^+ \cup n^+$
$\equiv (m \ominus \mathit{lbo}(m))^+ \cup [\downarrow m \mapsto \mathit{first}(n)] \cup$
$(m \ominus \mathit{lbo}(m))^+ \cup [\downarrow m \mapsto \mathit{first}(n)]) \, n^+ \cup n^+$
$\equiv (m \ominus \mathit{lbo}(m))^+ \cup [\downarrow m \mapsto \mathit{first}(n)] \cup$
$(m \ominus \mathit{lbo}(m))^+ \, n^+ \cup [\downarrow m \mapsto \mathit{first}(n)] n^+ \cup n^+$
$\equiv (m \ominus \mathit{lbo}(m))^+ \cup [\downarrow m \mapsto \mathit{first}(n)] \cup$
$[\downarrow m \mapsto \mathit{first}(n)] \, n^+ \cup n^+$.
Now $l^+ || n \equiv n^+ || n$ is immediate. Moreover,
$l^+ || \downarrow m$
$\equiv (m \ominus \mathit{lbo}(m))^+$
$\equiv m^+ || \downarrow m$,
so that (3) is satisfied. (2) is straightforward. (4) follows from (3) together with $p^+ n^+ \subseteq l$. Finally we show (1). Suppose $x \, l^+ \, x$ for some $x \in \downarrow m \cup \downarrow n$. Since $x \, p^+ \, n^+ \, x$ is not possible by $\downarrow p \cap \uparrow n \equiv \emptyset$, this would mean $x \, p^+ \, x$ or $x \, n^+ \, x$ contradicting $\mathit{isanchored}(m) \wedge \mathit{isanchored}(n)$. Linearity of l follows from (4) together with that of m^+ and n^+. Assume now $x \in \downarrow m$. Then $x \, p^* \, \mathit{lbo}(m) \, l \, \mathit{first}(m) \, n^+ \, \Box$ so that $\mathit{last}(l) \equiv \Box$.

4.2 A Version with Overwriting

Next, we want to derive a version of the function that works with header cells of chains within a larger storage rather than with the chains themselves. Moreover we overwrite the concatenated chain onto the part of the memory that contained the component chains. This can be specified as follows:

$$\begin{array}{rcl} \mathit{owconc}(l,x,y) & \overset{\text{def}}{\equiv} & \mathit{concpc}(\mathit{from}(x,l),\mathit{from}(y,l)) \rhd \\ & & l \Leftarrow \mathit{conc}(\mathit{from}(x,l),\mathit{from}(y,l)) \ . \end{array}$$

For the further development we assume $\mathit{concpc}(\mathit{from}(x,l),\mathit{from}(y,l)) \equiv \mathsf{true}$ and set

$$\begin{array}{rcl} m & \overset{\text{def}}{\equiv} & \mathit{from}(x,l) \ , \\ n & \overset{\text{def}}{\equiv} & \mathit{from}(y,I) \ . \end{array}$$

Then m and n by definition are restrictions of l so that we have $m, n \subseteq l$. Now we can simplify:

$$\begin{array}{l} \quad l \Leftarrow \mathit{conc}(m,n) \\ \equiv l \Leftarrow (m \ominus \mathit{lbo}(m) \cup [\mathit{lbo}(m) \mapsto \mathit{first}(m)] \cup n) \\ \equiv \text{(by Corollary 3.8)} \\ \quad l \Leftarrow (m \ominus \mathit{lbo}(m) \cup [\mathit{lbo}(m) \mapsto y] \cup n) \\ \equiv \text{(annihilation, commutativity of } \cup \text{)} \\ \quad l \Leftarrow [\mathit{lbo}(m) \mapsto y] \end{array}$$

Next we introduce an auxiliary function for computing this expression:

$$\mathit{owlbo}(l,x,y) \overset{\text{def}}{\equiv} l \Leftarrow [\mathit{lbo}(\mathit{from}(x,l)) \mapsto y] \ .$$

We have

$$\mathit{owconc}(l,x,y) \equiv \mathit{owlbo}(l,x,y) \ .$$

Now we derive a recursion equation for *owlbo*. Let again $m \equiv \mathit{from}(x,l)$. Then

$$\begin{array}{l} \quad \mathit{owlbo}(l,x,y) \\ \equiv \text{(unfold } \mathit{owlbo}\text{)} \\ \quad l \Leftarrow [\mathit{lbo}(m) \mapsto y] \\ \equiv \text{(by the recursion for } \mathit{lbo}\text{)} \\ \quad \mathsf{if}\ m(\mathit{first}(m)) = \square\ \mathsf{then}\ l \Leftarrow [\mathit{first}(m) \mapsto y] \\ \qquad\qquad\qquad\qquad\quad \mathsf{else}\ \ l \Leftarrow [\mathit{lbo}(\mathit{rest}(m)) \mapsto y]\ \mathsf{fi} \\ \equiv \text{(by Corollary 3.8)} \\ \quad \mathsf{if}\ l(x) = \square\ \mathsf{then}\ l \Leftarrow [x \mapsto y] \\ \qquad\qquad\quad \mathsf{else}\ \ l \Leftarrow [\mathit{lbo}(\mathit{from}(l(x),l)) \mapsto y]\ \mathsf{fi} \\ \equiv \text{(fold } \mathit{owlbo}\text{)} \end{array}$$

```
if $l(x) = \Box$ then $j \mathbin{\triangleleft\!\!-} [x \mapsto y]$
                 else $owlbo(l, l(x), y)$ fi .
```

Termination of this recursion is obvious. It is quite reassuring that the fundamental unfold/fold technique for deriving recursions also applies to pointer algorithms in this setting.

Since we have even obtained a tail-recursive version, we are already very close to an imperative program. To get there, we introduce a procedure specified by

```
proc $powconc \equiv$ (var state $l$, cell$x, y$) :
    $concpc(l, \mathit{from}(x, l), \mathit{from}(y, l)) \rhd$
    $l := owlbo(l, x, y)$
```

Note that this clearly specifies l as a transient parameter, whereas x and y are passed by value. Therefore the imperative version of *powconc* needs local variables for x and$y$, whereas it may operate on l directly. This is described by the following schematic rule for passing from a procedure that calls a tail-recursive function to a procedure with a loop in its body:

```
proc  $p \equiv$ (var m $a$,  n $b$) :
    $P(a, b) \rhd a := f(a, b)$
where
funct $f \equiv$ (m $a$, n $b$) m :
    if $C(a, b)$ then $T(a, b)$ else $f(K(a, b), L(a, b))$ fi
—————————————————— $\updownarrow$ ——————————————————[ NEW⟦$B$⟧
proc $p \equiv$ (var m $a$,  n $B$) :
    $P(a, B) \rhd$
    ⌈ var n $b := B$ ;
      while $\neg C(a, b)$ do $(a, b) := (K(a, b), L(a, b))$ od;
      $a := T(a, b)$                                  ⌋  .
```

Note that a, b, and B may stand for tuples of variables. The condition NEW⟦B⟧ states that B has to be a (tuple of) fresh identifier(s). Applying this rule we obtain

```
proc $powconc \equiv$ (var state $l$, cell $X, Y$) :
    $concpc(l, \mathit{from}(X, l), \mathit{from}(Y, l)) \rhd$
    ⌈ (var cell $x, y$) := $(X, Y)$ ;
      while $l(x) \neq \Box$ do $(l, x, y) := (l, l(x), y)$ od;
      $l := l \mathbin{\triangleleft\!\!-} [x \mapsto y]$                          ⌋
```

Our final version results from eliminating useless assignments of the form $z := z$ as well as the variable y which never is changed:

```
proc $powconc \equiv$ (var state $m$, cell $X, Y$) :
    $concpc(l, from(X, l), from(Y, l)) \rhd$
    $\lceil$ var cell $x := X$ ;
      while $l(x) \neq \Box$ do $x := l(x)$ od;
      $l := l \Leftarrow [x \mapsto Y]$                    $\rfloor$ .
```

If we write the assignment

$$l := l \Leftarrow [x \mapsto Y]$$

in a Pascal-like way as

$$x \uparrow := Y \ ,$$

(where l now is an implicit parameter), we see that we actually have derived a version with selective updating.

In the derivation we have not made use of any assumptions about absence of sharing. Indeed, if in l there are pointers from other data structures to (parts of) the lists headed by x and y, there will be indirect side effects on these pointers. However, since by the specification we know the value of the complete store after execution of our procedure, we can *calculate* these effects using our algebraic laws. Also, one can easily write stronger preconditions that exclude sharing if this is desired.

5 Chain Reversal

5.1 Specification and First Explicit Solution

Next we want to derive a procedure for reversing a non-empty chain "in situ". Again we first specify a purely applicative version. The reverse of a chain should contain exactly the same proper cells as the original chain, however, in reverse order of traversal.

We can express this as follows:

$$\begin{array}{rcl} revpc(m) & \overset{\text{def}}{\equiv} & m \not\equiv \emptyset \wedge isanchored(m), \\ \\ rev(m) & \overset{\text{def}}{\equiv} & revpc(m) \rhd \textsf{some state } n : \\ & (1) & isanchored(n) \wedge \\ & (2) & \downarrow n = \downarrow m \wedge \\ & (3) & n^{+} || \downarrow m = (m^{+})^{-1} || \downarrow m \ . \end{array}$$

Here, $(m^+)^{-1}$ is the converse of the relation m^+.

Let us now derive an explicit form of $\mathit{rev}(m)$. First we observe that

$$(m^+)^{-1} \equiv (m^{-1})^+ ,$$

where m^{-1} again is a map, since by Lemma 3.1 m is injective. Hence (3) can be satisfied if we can choose n in such a way that

$$n||{\downarrow}m \equiv m^{-1}||{\downarrow}m .$$

Therefore we now first calculate $m^{-1}||{\downarrow}m$. We have, using Lemmas 3.5 and 3.6, that

$$\begin{aligned}
& {\downarrow}m \\
\equiv\ & {\uparrow}m\backslash\{\Box\} \cup \{\mathit{first}(m)\} \\
\equiv\ & {\downarrow}m^{-1}\backslash\{\Box\} \cup \{\mathit{first}(m)\} \\
\equiv\ & {\downarrow}(m^{-1} \ominus \Box) \cup \{\mathit{first}(m)\} .
\end{aligned}$$

Moreover,

$$m^{-1}||{\downarrow}m \equiv m^{-1}|{\downarrow}m .$$

Now,

$$\begin{aligned}
& m^{-1}||{\downarrow}m \\
\equiv\ & m^{-1}|{\downarrow}m \\
\equiv\ & m^{-1}|({\downarrow}(m^{-1} \ominus \Box) \cup \{\mathit{first}(m)\}) \\
\equiv\ & m^{-1}|{\downarrow}(m^{-1} \ominus \Box) \cup m^{-1}|\{\mathit{first}(m)\}) \\
\equiv\ & m^{-1} \cup m^{-1}|\{\mathit{first}(m)\} \\
\equiv\ & m^{-1} \ominus \Box \cup \emptyset \\
\equiv\ & m^{-1} \ominus \Box .
\end{aligned}$$

Since we need to have ${\downarrow}n \equiv {\downarrow}m$ we now try to find a z such that

$$n \equiv m^{-1} \ominus \Box \cup [\mathit{first}(m) \mapsto z] .$$

We also need to satisfy $\mathit{isanchored}(n)$ which implies $\Box \in {\uparrow}n$. However,

$${\uparrow}(m^{-1} \ominus \Box) \subseteq {\uparrow}m^{-1} \subseteq {\downarrow}m$$

and hence $\Box \notin {\uparrow}(m^{-1} \ominus \Box)$. So the only way to achieve $\Box \in {\uparrow}n$ is to set $z \equiv \Box$. This leaves us with

$$(*) \quad n \equiv m^{-1} \ominus \Box \cup [\mathit{first}(m) \mapsto \Box] .$$

Now one can verify $\mathit{isanchored}(n)$. This will be the starting point for our development of an imperative version.

One can bring the expression for n into a different form. We have

$$\begin{aligned}
&m^{-1} \ominus \Box \cup [\mathit{first}(m) \mapsto \Box] \\
\equiv\ & m^{-1} \ominus \Box \cup [\mathit{first}(m) \mapsto \Box] \ominus \Box \\
\equiv\ & (m^{-1} \cup [\mathit{first}(m) \mapsto \Box]) \ominus \Box \\
\equiv\ & (m^{-1} \cup [\Box \mapsto \mathit{first}(m)]^{-1}) \ominus \Box \\
\equiv\ & (m \cup [\Box \mapsto \mathit{first}(m)])^{-1} \ominus \Box .
\end{aligned}$$

This form now suggests a nice conceptual algorithm for reversing a chain: First close the chain temporarily to a cycle by treating $\Box$as an ordinary cell and adding the link $[\Box \mapsto \mathit{first}(m)]$. Then reverse all the pointers by passing to $(m \cup [\Box \mapsto \mathit{first}(m)])^{-1}$. Finally, cut the cycle again by taking away the link emanating from $\Box$. It should be remarked here that this view of chain reversal was not known to the author before; it was obtained by purely symbolic reasoning in trying to bring the original expression into a more regular form.

5.2 Versions with Overwriting and Selective Updating

Again we specify a version of reversal that deals with lists addressed by header cells:

$$\begin{aligned}
\mathit{owrev}(l,x) \quad \overset{\text{def}}{\equiv} \quad & \mathit{revpc}(\mathit{from}(x,l)) \rhd \\
& l \mathbin{\triangleleft\!\!\!+} \mathit{rev}(\mathit{from}(x,l)) \ .
\end{aligned}$$

Using $(*)$ we obtain

$$\begin{aligned}
&\mathit{owrev}(lx) \\
\equiv\ & l \mathbin{\triangleleft\!\!\!+} (\mathit{from}(x,l)^{-1} \ominus \Box \cup [\mathit{first}(\mathit{from}(x,l)) \mapsto \Box]) \\
\equiv\ & l \mathbin{\triangleleft\!\!\!+} (\mathit{from}(x,l)^{-1} \ominus \Box \cup [x \mapsto \Box]) \\
\equiv\ & \text{(commutativity, sequentialization)} \\
& l \mathbin{\triangleleft\!\!\!+} [x \mapsto \Box] \mathbin{\triangleleft\!\!\!+} \mathit{from}(x,l)^{-1} \ominus \Box .
\end{aligned}$$

Now we introduce an auxiliary function

$$\mathit{owrev1}(l,x,y) \overset{\text{def}}{\equiv} l \mathbin{\triangleleft\!\!\!+} [x \mapsto y] \mathbin{\triangleleft\!\!\!+} \mathit{from}(x,l)^{-1} \ominus \Box$$

with the embedding

$$\mathit{owrev}(l,x) \equiv \mathit{owrev1}(l,x,\Box) \ .$$

For abbreviation we introduce $m \overset{\text{def}}{\equiv} l \mathbin{\triangleleft\!\!\!+} [x \mapsto y]$. Now we can develop a recursion equation:

$$\begin{aligned}
&\mathit{owrev1}(l,x,y) \\
\equiv\ & \text{(unfold } \mathit{owrev}) \\
& l \mathbin{\triangleleft\!\!\!+} [x \mapsto y] \mathbin{\triangleleft\!\!\!+} \mathit{from}(x,l)^{-1} \ominus \Box \\
\equiv\ & m \mathbin{\triangleleft\!\!\!+} \mathit{from}(x,l)^{-1} \ominus \Box
\end{aligned}$$

```
  ≡ (by Corollary 3.8)
    if $l(x) = \Box$ then $m \Lleftarrow [x \mapsto \Box]^{-1} \ominus \Box$
                 else $m \Lleftarrow ([x \mapsto l(x)] \cup \mathit{from}(l(x), l))^{-1} \ominus \Box$ fi
  ≡ (distributivity, inverse)
    if $l(x) = \Box$ then $m \Lleftarrow [\Box \mapsto x] \ominus \Box$
                 else $m \Lleftarrow ([l(x) \mapsto x] \ominus \Box \cup \mathit{from}(l(x), l)^{-1} \ominus \Box)$ fi
  ≡ ($l(x) \neq \Box$ in else-case)
    if $l(x) = \Box$ then $m \Lleftarrow \emptyset$
                 else $m \Lleftarrow ([l(x) \mapsto x] \cup \mathit{from}(l(x), l)^{-1} \ominus \Box)$ fi
  ≡ (neutrality, sequentialization)
    if $l(x) = \Box$ then $m$
                 else $m \Lleftarrow ([l(x) \mapsto x] \cup \mathit{from}(l(x), l)^{-1} \ominus \Box$ fi
  ≡ (Lemma 3.4)
    if $l(x) = \Box$ then $m$
                 else $m \Lleftarrow [l(x) \mapsto x] \Lleftarrow \mathit{from}(l(x), l)^{-1} \ominus \Box$ fi
  ≡ (fold $\mathit{owrev1}$)
    if $l(x) = \Box$ then $m$
                 else $\mathit{owrev1}(m, l(x), x)$ fi .
```

Again we have arrived at an (obviously terminating) tail recursion.

Specifying a procedure

```
proc $\mathit{powrev} \equiv$ (var state $l$, cell $X$) :
    $\mathit{revpc}(\mathit{from}(X, l)) \rhd$
    $l := \mathit{rev}(\mathit{from}(X, l))$
```

we obtain, as in the previous section, the final version

```
proc $\mathit{powrev} \equiv$ (var state $m$, cell $X$) :
    $\mathit{revpc}(\mathit{from}(X, l)) \rhd$
    ⌈ (var cell $x, y$) := $(X, \Box)$ ;
      while $l(x) \neq \Box$
         do $(l, x, y) := (l \Lleftarrow [x \mapsto y], l(x), x)$ od ;
           $l := l \Lleftarrow [x \mapsto y]$                          ⌋
```

Note that sequentialization of the collective assignment would require an auxiliary variable.

This program describes a well-known algorithm for reversing a list “in situ”. Whereas verification purely at the procedural level is by no means easy (see e.g. [4,7]), in particular if all the details were to be filled in, we have derived and thereby verified the program by a fairly short and simple formal calculation using standard transformation techniques.

5.3 Proving Program Properties

To conclude this section, we want to show how the algebra of maps can also be used for proving properties of programs. We want to show that, for a non-empty finite anchored chain m,

$$rev(rev(m)) \equiv m \; .$$

There is a variety of ways of showing this; one possibility would be an induction on the length of m using a recursive or imperative variant of *rev*. However, the proof is much simpler if one uses the non-operational explicit form $(*)$ we have derived from the implicit specification, since induction then can completely be avoided. We first note that

$$\begin{aligned} & rev(m) \\ \equiv\; & m^{-1} \ominus \square \cup [\mathit{first}(m) \mapsto \square] \\ \equiv\; & (m \ominus (\square{\downarrow}m))^{-1} \cup [\mathit{first}(m) \mapsto \square] \; . \end{aligned}$$

Next,

$$\begin{aligned} & \{\mathit{first}(rev(m))\} \\ \equiv\; & {\downarrow}rev(m) \backslash {\uparrow}rev(m) \\ \equiv\; & {\downarrow}m \backslash ({\uparrow}(m \ominus (\square{\downarrow}m))^{-1} \cup \{\square\}) \\ \equiv\; & {\downarrow}m \backslash \{\square\} \backslash {\downarrow}(m \ominus (\square{\downarrow}m)) \\ \equiv\; & {\downarrow}m \backslash {\downarrow}(m \ominus (\square{\downarrow}m)) \\ \equiv\; & {\downarrow}m \backslash ({\downarrow}m \backslash (\square{\downarrow}m)) \\ \equiv\; & {\downarrow}m \cap \square{\downarrow}m \\ \equiv\; & \square{\downarrow}m \; . \end{aligned}$$

Now

$$\begin{aligned} & rev(rev(m)) \\ \equiv\; & rev(m)^{-1} \ominus \square \cup [\mathit{first}(rev(m)) \mapsto \square] \\ \equiv\; & ((m \ominus (\square{\downarrow}m))^{-1} \cup [\mathit{first}(m) \mapsto \square])^{-1} \ominus \square \cup \\ & [\square{\downarrow}m \mapsto \square] \\ \equiv\; & (((m \ominus (\square{\downarrow}m))^{-1})^{-1} \cup [\mathit{first}(m) \mapsto \square]^{-1}) \ominus \square \cup \\ & [\square{\downarrow}m \mapsto \square] \\ \equiv\; & (m \ominus (\square{\downarrow}m) \cup [\square \mapsto \mathit{first}(m)]) \ominus \square \cup \\ & [\square{\downarrow}m \mapsto \square] \\ \equiv\; & (m \ominus (\square{\downarrow}m)) \ominus \square \cup [\square \mapsto \mathit{first}(m)]) \ominus \square \cup \\ & [\square{\downarrow}m \mapsto \square] \\ \equiv\; & m \ominus (\square{\downarrow}m) \cup [\square{\downarrow}m \mapsto \square] \\ \equiv\; & m \; . \end{aligned}$$

In a similar fashion one can prove the associativity of *conc*. Since these proofs manage to do completely without induction, their mechanization should be easy, e.g., using techniques from term rewriting. A tool for assisting such proofs would considerably help our approach in becoming practically usable.

6 Conclusion

We have shown with two examples how to derive algorithms involving pointers and selective updating from formal specifications using standard transformation techniques. The key to the method consists in considering the store as an explicit parameter, since then one has complete information about sharing and therefore complete control about side effects. We deem this approach much clearer (and much more convenient) than the idea of hiding the store and coming up with special logics (see e.g. [8,6]) that capture the side-effects indirectly, as needs to be done in the field of verification of procedural programs.

Staying at the applicative level almost to the very end of the derivations has allowed us to take full advantage of the powerful algebra of partial maps. The operations of that algebra are even that expressive that we did not need to explain anything with the help of diagrams. This may seem due to the simplicity of the algorithms. However, also when developing the intricate garbage collection algorithm described in [3] we quite soon stopped drawing diagrams because the algebraic formulation was clearer and much more modular. Another advantage of the applicative treatment is that if additional predicates or operations on maps are needed, they are much more easily added at the applicative than at the procedural level. Finally, if pointer algorithms are developed in a systematic way at the applicative language level, there is no need for introducing additional imperative language concepts such as the highly imperspicuous pointer rotation [12].

We are convinced that our approach can be extended into a convenient method for constructing systems software with guaranteed correctness.

Acknowledgements
The idea of an algebraic treatment of pointers was stimulated by discussions within IFIP WG 2.1, notably by the algebraic way in which R. Bird and L. Meertens develop tree and list algorithms. I gratefully acknowledge many helpful conversations with my present and former colleagues from the project CIP, notably with F.L. Bauer, ∪. Berger, H. Partsch, P. Pepper, W. Meixner, and, particularly, H. Ehler, as well as with my students.

References

1. F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T.A. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, H. Wössner: The Munich project CIP. Volume I: The wide spectrum language CIP-L. Lecture Notes in Computer Science **183**. Berlin: Springer 1985
2. F.L. Bauer, B. Möller, H. Partsch, P. Pepper: Formal program construction by transformations — Computer-aided, Intuition-guided Programming. Institut für Informatik der T$\cup$München, TUM-I8807, Juni 1988. Also in IEEE Transactions on Software Engineering **15**, 165–180 (1989)
3. U. Berger, W. Meixner, B. Möller: Calculating a garbage collector. In: M. Broy M. Wirsing (ed.): Methodik des Programmierens. Fakultät für Mathematik und Informatik der Universität Passau, MIP-8915, 1989, 1–52. Also in: M. Broy, M. Wirsing (eds.): Programming methodology — The CIP approach. To appear in Lecture Notes in Computer Science . Berlin: Springer
4. R. Burstall: Some techniques for proving correctness of programs which alter data structures. In: B. Meltzer, D. Mitchie (eds.): Machine Intelligence **7**. Edinburgh University Press 1972, 23–50
5. A. Horsch: Functional programming with partially applicable operators. Fakultät für Mathematik und Informatik der TU München, Dissertation, 1989
6. A. Kausche: Modale Logiken von geflechtartigen Datenstrukturen und ihre Kombination mit temporaler Programmlogik. Fakultät für Mathematik und Informatik der T$\cup$München, Dissertation, 1989
7. M. Levy: Verification of programs with data referencing. Proc. 3me Colloque sur la Programmation 1978, 413–426
8. I. Mason: Verification of programs that destructively manipulate data. Science of Computer Programming **10**, 177–210 (1988)
9. B. Möller: Applicative assertions. In: J.L.A. van de Snepscheut (ed.): Mathematics of Program Construction. Lecture Notes in Computer Science 375. Berlin: Springer 1989, 348–362
10. P. Pepper, B. Möller: Programming with (finite) mappings. In: M. Broy (ed.): Informatik im Kreuzungspunkt von Numerischer Mathematik, Rechnerentwurf, Programmierung, Algebra und Logik. Festkolloquium für F.L. Bauer, Juni 1989.
11. J. Reynolds: Reasoning about arrays. Commun. ACM **22**, 290–299 (1979)
12. N. Suzuki: Analysis of pointer rotation. Conf. Record 7th POPL, 1980, 1–11. Revised version: Commun. ACM **25**, 330–335 (1982)